# Coding Challenge Dokumentation

## Inhalt 🚚

1. Analyse der Problemstellung
2. Ungeeignete Lösungsansätze
3. Eigener Algorithmus zur Lösung
4. Ausführung des Lösungsprogramms
5. Lösung

## 1. Analyse der Problemstellung

Bei der Aufgabenstellung handelt es sich um eine Variante des Knapsack Problems, welches ein Optimierungsproblem darstellt in dem man versucht Gegenstände, welche einen Wert und ein Gewicht haben, so in einen Rucksack mit begrenzter Traglast zu laden, dass der Gesamtwert aller Gegenstände im Rucksack maximal ist.
Noch genauer handelt es sich dabei um das Multiple Knapsack Problem, da die Gegenstände auf mehrere Transporter aufgeteilt werden sollen.

## 2. Ungeeignete Lösungsansätze

### 2.1 Bestehende Lösungsmethoden speziell für das Knapsack-Problem

Da zum Knapsack-Problem bereits Lösungsmethoden bestehen, könnten diese auf die Aufgabenstellung angepasst und angewandt werden, um die Berechnung einer optimalen Lösung zu garantieren. Da bestehende Lösungsalgorithmen allerdings nicht für so große Mengen an Gegenständen wie in dieser Aufgabenstellung ausgelegt sind, erschien mir dies aus folgenden Gründen nicht der richtige Lösungsweg zu sein:

1. Bei bestehenden Algorithmen wird über die Kapazitäten aller Rucksäcke und die Menge der Gegenstände iteriert. Da in der Aufgabenstellung allein die Größe eines Transporters mehr als 1 Mio Gramm beträgt, lässt sich bei nur zwei Transportern schon eine Gesamtzahl von mehreren Billionen Iterationen errechnen.
2. Betrachtet man die Tatsache, dass die Zwischenergebnisse der einzelnen Iterationen in einer mehrdimensionalen Matrix zwischengespeichert werden, so kann man sich errechnen, dass ohne signifikante Optimierungen mehrere TB Arbeitsspeicher benötigt werden, was die Kapazitäten jedes normalen Computers um weiten übersteigt.
3. Die Speicherauslastung kann zwar optimiert werden, allerdings ist auch die Laufzeit, welche sich bei dem Ausmaß der gegebenen Aufgabenstellung auf mehrere Stunden bis Tage beläuft, ein grundlegendes Problem.

Aufgrund der Abwägung zwischen einem sehr geringen Risiko, dass das Ergebnis nicht zu 100% optimal ist und einer extrem hohen Berechnungszeit habe ich mich daher für erstere Möglichkeit entschieden, da beim Anwendungsgebiet, einen Transporter zu beladen auch in Realität eine Toleranz von wenigen Gramm unproblematisch sein dürfte.

### 2.2 Linear Programming

Die Aufgabenstellung könnte als lineares Gleichungssystem mit Constraints und einem Wert, welcher maximal werden soll, ausgedrückt werden. Mithilfe von Lösungsbibliotheken wie zum Beispiel dem *Glop Linear Solver* der *Google OR-Tools* könnte dann eine möglichst optimale Lösung schnell bestimmt werden. Aufgrund folgender Gründe habe ich mich allerdings trotzdem auch gegen diesen Ansatz entschieden:

1. Durch die Benutzung von entsprechenden Solver-Bibliotheken entstehen viele Abhängigkeiten meines Codes von anderen Code-Stücken und teilweise sogar von auf dem Computer installierten Programmen.

2. Da man bei diesem Lösungsansatz lediglich die Aufgabenstellung umformulieren würde und die eigentliche Lösung des Problems dem bereits vorhandenen Solver überlässt, denke ich, dass dies nicht das Ziel des Wettbewerbs ist, da kaum Eigenleistung erbracht werden müsste.

## 3. Eigener Algorithmus zur Lösung

Aufgrund der soeben diskutierten schlechten Eignung der vorhandenen Lösungsmöglichkeiten habe ich den folgenden eigenen Lösungsalgorithmus entwickelt.

### 3.1 Funktionsweise

1. Gegenstände aus der Datei `items.csv` und Transporter aus der Datei `trucks.csv` einlesen
2. Gegenstände nach Effizienz (Nutzwert/Gewicht) sortieren
3. Alle Transporter mit den effizientesten Gegenständen befüllen. Wenn keine mehr übrig sind oder nicht mehr genug Platz ist, mit zweit-effizientesten Gegenständen weitermachen und so weiter
4. Gegenstands-Gruppen zwischen je zwei Transportern versuchen so zu tauschen, dass die freie Kapazität in einem Transporter maximal wird, um in einem Transporter maximalen Platz für weitere Hardware zu schaffen
5. Für jeden Transporter versuchen, Gegenstands-Gruppen mit Gegenstands-Gruppen aus dem übrigen nicht verladenen Vorrat so zu tauschen, dass der Gesamt-Nutzwert der Gegenstände im Transporter steigt
6. Falls in *4.* oder *5.* eine Änderung durchgeführt wurde, wieder zu *4.* springen und dort fortsetzen
7. Die Finale Beladung der Transporter in der Datei `solution.csv` speichern und anschließend ausgeben

> Es werden bei Gegenstands-Gruppen immer nur alle möglichen Gruppen der Menge 1 bis 5 betrachtet, um die Laufzeit ohne signifikante Qualitätsverluste erheblich zu reduzieren.

> Für die gegebene Aufgabenstellung findet der Algorithmus eine optimale Lösung

### 3.2 Vorteile des eigenen Algorithmus

1. Die Laufzeit beläuft sich nur auf wenige Sekundenbruchteile bis Sekunden
2. Es werden für Gewichte und Nutzwerte auch Kommazahlen akzeptiert, was mit vielen anderen Lösungsalgorithmen nur sehr schwer umsetzbar wäre
3. Die Eingabewerte sind frei anpassbar und erweiterbar. So können zum Beispiel zur Eingabedatei `trucks.csv` einfach weitere Transporter hinzugefügt werden, falls eine Ladeliste für mehr als zwei Transporter benötigt wird
4. Es werden keinerlei Bibliotheken als Abhängigkeiten benötigt, das heißt, um das Programm auszuführen, muss lediglich Java JDK 8 installiert sein

## 4. Ausführung des Lösungsprogramms

### 4.1 Voraussetzungen

#### 4.1.1 Empfehlung

Java SE Development Kit 8 (https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html) muss installiert sein (da JavaFX in der Version integriert ist, ist diese zwingend erforderlich, falls die Benutzeroberfläche verwendet werden soll)

> Die Ausführung unter Windows wird empfohlen

#### 4.1.2 Optionen unter Verzicht auf die grafische Benutzeroberfläche

Falls die Darstellung in der Konsole ausreicht (etwas weniger übersichtliche Darstellung und Verzicht auf Visualisierung der Optimierungsvorgänge), kann ein beliebiges Java Development Kit, wie zum Beispiel OpenJDK oder AdoptOpenJDK, verwendet werden, solange die Version mindestens 8 ist. Anschließend muss beim Starten des Programms die `_nogui` -Version gewählt werden.

> **Wichtig:** Ein JRE (Java Runtime Environment) reicht nicht aus, da damit der Code nicht kompiliert werden kann

OpenJDK 8 JDK Installation unter Linux:

- Debian, Ubuntu, etc.: `sudo apt-get install openjdk-8-jdk`
- Fedora, Oracle Linux, Red Hat Enterprise Linux, etc.: `su -c "yum install java-1.8.0-openjdk-devel"`

AdoptOpenJDK 8 JDK Installation unter Windows:

1. Downloadseite öffnen (https://adoptopenjdk.net/releases.html?variant=openjdk8&jvmVariant=hotspot)
2. Richtige Windows-Version (*x86* oder *x64*) auswählen und die Installationsdatei (*.msi*) für **JDK** herunterladen
3. Die Installationsdatei ausführen und den Anweisungen folgen

## 4.2 Ausführung

#### 4.2.1 Windows

1. Im Stammverzeichnis des Repositories das Skript `start.bat` ( `start_nogui.bat` , falls ohne Benutzeroberfläche gestartet werden soll) lokalisieren
2. Das Skript entweder per Doppelklick oder aus der Konsole mit dem Befehle `.\start.bat` ( `.\start_nogui.bar` , falls ohne Benutzeroberfläche gestartet werden soll) ausführen
3. Das Programm wird nun kompiliert um anschließend automatisch ausgeführt
4. Die Lösung erscheint nach der Berechnung in der Konsole und in der sich wahlweise öffnenden Benutzeroberfläche. Außerdem wird sie in der Datei `solution.csv` gespeichert

#### 4.2.2 Linux

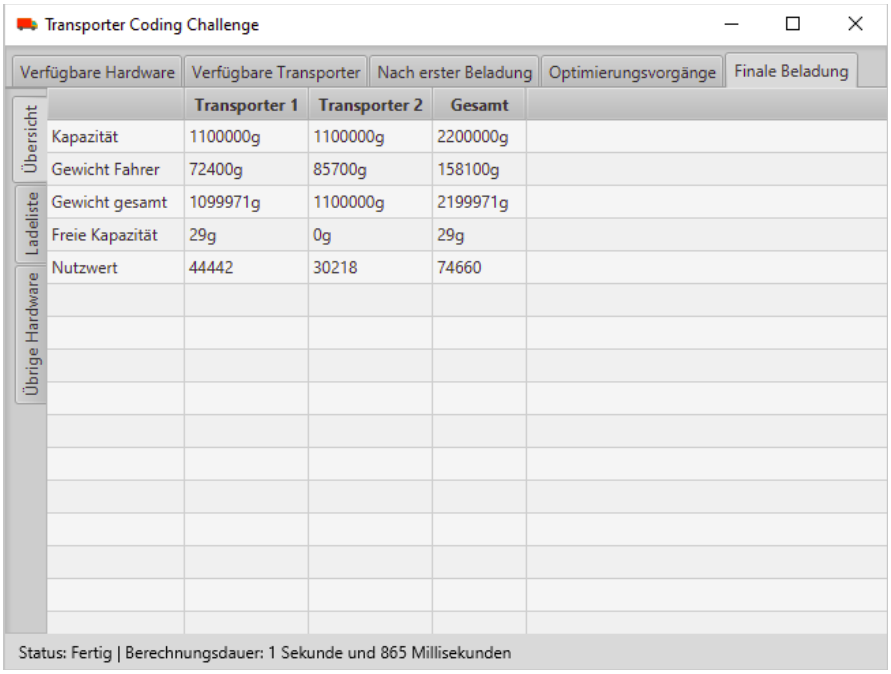
1. Im Stammverzeichnis des Repositories das Skript `start.sh` ( `start_nogui.sh` , falls ohne Benutzeroberfläche gestartet werden soll) lokalisieren
2. Das Skript aus der Konsole mit dem Befehle `sh ./start.sh` ( `sh ./start_nogui.sh` , falls ohne Benutzeroberfläche gestartet werden soll) ausführen
3. Das Programm wird nun kompiliert um anschließend automatisch ausgeführt
4. Die Lösung erscheint nach der Berechnung in der Konsole und in der sich wahlweise öffnenden Benutzeroberfläche. Außerdem wird sie in der Datei `solution.csv` gespeichert

## 4.3 Benutzeroberfläche

Die Benutzeroberfläche visualisiert den Verlauf des Algorithmus und die Lösung. Dazu ist sie in fünf tabs untergliedert:

- **Verfügbare Hardware**: Listet die zu verladende Hardware auf
- **Verfügbare Transporter**: Listet die verfügbaren Transporter auf
- **Nach erster Beladung**: Zeigt den Zustand der Beladung nach Schritt *3.*, vor die Optimierung stattgefunden hat
- **Optimierungsvorgänge**: Visualisiert die getätigten Aktionen der Schritte *4.* bis *6.*, um den Gesamt-Nutzwert zu maximieren
- **Finale Beladung**: Zeigt den Finalen Zustand der Beladung nach der ausführung des gesamten Algorithmus und damit die Lösung

> In den Tabs **Nach erster Beladung** und **Finale Beladung** kann die Ansicht jeweils zwischen *Übersicht*, *Ladeliste* und *Übrige Hardware* gewechselt werden, um spezifische Informationen zu erhalten




## 5. Lösung

### 5.1 Optimale Verteilung der Hardware

| Hardware | Einheiten Transporter 1 | Einheiten Transporter 2 | Einheiten Gesamt |
|---|---|---|---|
| Mobiltelefon Outdoor | 152 | 5 | 157 |
| Mobiltelefon Heavy Duty | 214 | 6 | 220 |
| Mobiltelefon Büro | 60 | 0 | 60 |
| Tablet outdoor groß | 284 | 86 | 370 |
| Tablet Büro klein | 3 | 592 | 595 |
| Tablet Büro groß | 0 | 0 | 0 |
| Tablet outdoor klein | 4 | 0 | 4 |
| Notebook outdoor | 0 | 0 | 0 |
| Notebook Büro 13" | 0 | 0 | 0 |
| Notebook Büro 14" | 0 | 0 | 0 |

### 5.2 Gesamtwerte

| | Transporter 1 | Transporter 2 | Gesamt |
|---|---|---|---|
| **Kapazität** | 1100000g | 1100000g | 2200000g |
| **Gewicht Fahrer** | 72400g | 85700g | 158100g |
| **Gewicht gesamt** | 1099971g | 1100000g | 2199971g |
| **Freie Kapazität** | 29g | 0g | 29g |

| | Transporter 1 | Transporter 2 | Gesamt |
|---|---|---|---|
| **Nutzwert** | 44442 | 30218 | **74660** |

Summe aller Nutzwerte: **`74660`**

### 5.3 Nicht verladene Hardware

| Nicht verladene Hardware | Übrige Einheiten |
|---|---|
| Mobiltelefon Outdoor | 0 |
| Mobiltelefon Heavy Duty | 0 |
| Mobiltelefon Büro | 0 |
| Tablet outdoor groß | 0 |
| Tablet Büro klein | 25 |
| Tablet Büro groß | 250 |
| Tablet outdoor klein | 536 |
| Notebook outdoor | 450 |
| Notebook Büro 13" | 205 |
| Notebook Büro 14" | 420 |